Int J Parallel Prog
DOI 10.1007/s10766-013-0252-y

Boosting CUDA Applications with CPU–GPU Hybrid Computing

Changmin Lee · Won Woo Ro · Jean-Luc Gaudiot

Received: 10 September 2012 / Accepted: 14 May 2013

Abstract This paper presents a cooperative heterogeneous computing framework which enables the efficient utilization of available computing resources of host CPU cores for CUDA kernels, which are designed to run only on GPU. The proposed system exploits at runtime the coarse-grain thread-level parallelism across CPU and GPU, without any source recompilation. To this end, three features including a work distribution module, a transparent memory space, and a global scheduling queue are described in this paper. With a completely automatic runtime workload distribution, the proposed framework achieves speedups of 3.08× in the best case and 1.42× on average compared to the baseline GPU-only processing.

Keywords Heterogeneous computing · Parallel processing · GPGPU · CUDA

1 Introduction

General-Purpose computing on Graphics Processing Units (GPGPU) has recently emerged as a powerful computing paradigm because of the massive parallelism provided by several hundreds of processing cores [5,18]. Under the GPGPU concept, NVIDIA has developed a C-based programming model, Compute Unified Device

C. Lee · W. W. Ro (✉)
Yonsei University, Seoul 120-749, Republic of Korea
e-mail: wro@yonsei.ac.kr

C. Lee
e-mail: exahz@yonsei.ac.kr

J.-L. Gaudiot
University of California, Irvine, CA 92697-2625, USA
e-mail: gaudiot@uci.edu

Published online: 22 May 2013

Architecture (CUDA), which provides greater programmability for high-performance graphics devices. As a matter of fact, general-purpose computing on graphics devices with CUDA helps improve the performance of many applications under the concept of a Single Instruction Multiple Thread model (SIMT).

Although the GPGPU paradigm successfully provides significant computation throughput, its performance could still be improved if we could utilize the idle CPU resource. Indeed, in general, the host CPU is being held while the CUDA kernel executes on the GPU devices; the CPU is not allowed to resume execution until the GPU has completed the kernel code and has provided the computation results. The main motivation of our research is to exploit parallelism across the host CPU cores in addition to the GPU cores. This will eventually provide additional computing power for the kernel execution while utilizing the idle CPU cores. Our ultimate goal is thus to provide a technique which eventually exploits sufficient parallelism across heterogeneous processors.

The paper proposes *Cooperative Heterogeneous Computing* (CHC), a new computing paradigm for explicitly processing CUDA applications in parallel on sets of heterogeneous processors including x86 based general-purpose multi-core processors and graphics processing units. There have been several previous research projects which have aimed at exploiting parallelism on CPU and GPU. However, those previous approaches require either additional programming language support or API development. As opposed to those previous efforts, our CHC is a software framework that provides a virtual layer for transparent execution over host CPU cores. This enables the direct execution of CUDA code, while simultaneously providing sufficient portability and backward compatibility.

To achieve an efficient cooperative execution model, we have developed three important techniques:

- A workload distribution module (WDM) for CUDA kernel to map each kernel onto CPU and GPU
- A memory model that supports a transparent memory space (TMS) to manage the main memory with GPU memory
- A global scheduling queue (GSQ) that supports balanced thread scheduling and distribution on each of the CPU cores

We present a theoretical analysis of the expected performance to demonstrate the maximum feasible improvement of our proposed system. In addition, the performance has been evaluated on a real system, and the results show that speedups as high as 3.08× (in the best case) could be achieved. On average, the complete CHC system shows a performance improvement of 1.42× over GPU-only computation with 14 CUDA applications.

The rest of the paper is organized as follows. Section 2 introduces the existing CUDA programming model and the related background. Section 3 describes a motivation and an experimental analysis of this work. In Sect. 4, we present the design and implementation of the CHC framework. Section 5 gives preliminary results for CUDA applications by adopting the CHC framework. Section 6 discusses the experimental results of our CHC framework and Sect. 7 summarizes the major differences between our system and related work. Finally, we conclude the work in Sect. 8.

2 Background

Originally designed as special hardware for real-time and high-definition 3D graphics, GPUs have evolved into many-core processors to accelerate highly parallel computations. The GPU is designed such that more transistors are devoted to processing cores rather than the sophisticated control hardware, and therefore able to address problems that can be represented as data-parallel computations.

Many applications with data-parallelism can map data elements to processing threads. For each data element we can only read from the input, execute some operations on it, and write to the output; it is possible to have multiple inputs and multiple outputs. For example, video processing applications such as video encoding and decoding can map image frames, blocks, and pixels to processing threads with dedicated inputs and outputs.

The CUDA technology [19] introduced by NVIDIA in 2006 has become the most effective solution for general-purpose parallel computing on GPUs. It is designed to leverage processing cores (or Streaming Multiprocessors, SMs) in NVIDIA GPUs to execute data-parallel functions, called kernels. To this end, CUDA provides a programming model to deliver an efficient way to express the kernels with a minimal language extension.

The CUDA programming model consists of three abstractions, including a thread hierarchy, a memory hierarchy, and barrier synchronization (as shown in Fig. 1). A CUDA thread is the smallest unit of the GPU processing, and therefore can provide fine-grained data parallelism and thread-level parallelism (TLP). During a kernel execution, a thread has per-thread local memory to store its private data, and they may access data from the shared memory or the global memory.

A block of threads can provide coarse-grained data parallelism. Each block of threads can be executed independently in parallel, so that this independence allows thread blocks to be scheduled in any order across any available SMs. In fact, threads within a block cooperate with each other by sharing data through shared memory, which is dedicated to an SM.

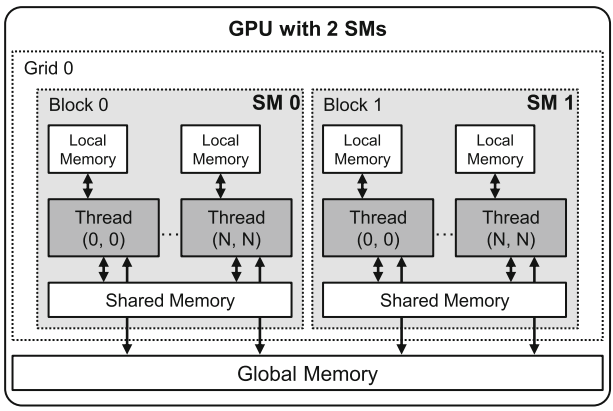

Fig. 1 The CUDA architecture: thread hierarchy and memory hierarchy

Synchronization between threads to control memory accesses is allowed by explicitly specifying synchronization points in the kernel. The synchronization point, called a barrier, keeps all threads in the block from proceeding to the next instruction until all other threads reach the barrier. It is necessary to prevent race conditions, and thus to ensure that all threads are reading the correct values from shared memory.

Based on these core abstractions, CUDA provides a heterogeneous programming model that enables to use the right core for the right job in order to achieve high performance.

3 Motivation

In the current heterogeneous computing models such as CUDA and OpenCL, CPU cores and GPU cores cooperate with each other (e.g., co-processing). They assume that a parallel portion of a code (i.e., CUDA kernels) runs on a GPU and a serial portion of a code (i.e., the rest of the C program) runs on a CPU [18]. This heterogeneity across different processing units is a similar concept to that of the heterogeneous chip multiprocessor which exploits both large cores and small cores to address a much wider spectrum of system workloads [13]. In CUDA applications, then, a critical question is how to structure a certain portion of codes to expose as much data parallelism. We have observed that CUDA applications are typically designed to leverage the massive parallelism only with GPUs. Exploiting the massive parallelism with host threads is not a concern for the program design, so that CPU cores are held in the idle state even at runtime. We believe that it is interesting to increase the performance of a CUDA application at a given time by using all the processing units.

3.1 Problem Observation

One of the major roles of the host CPU for the CUDA kernel is limited to controlling and accessing the graphics devices, while the GPU device provides a massive amount of data parallelism. Figure 2a shows an example where the host controls the execution

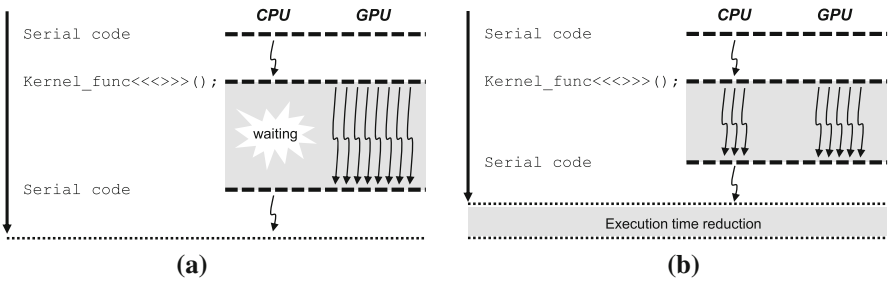

Fig. 2 Execution flow of CUDA program: **a** limitation where CPU stalls and **b** cooperative execution in parallel

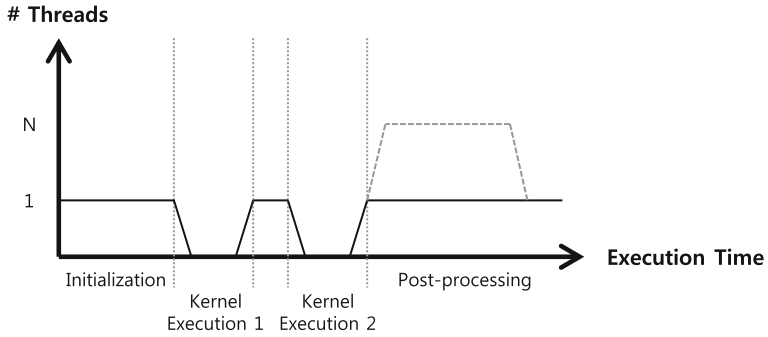

Fig. 3 Average CPU load during a CUDA application runtime

flow of the program only, while the device is responsible for executing the kernel. Once a CUDA program is started, the host processor executes the program sequentially until the kernel code is encountered. As soon as the host calls the kernel launch function, the device starts to execute the kernel with a large number of hardware threads on the GPU device. In fact, the host thread is held in the idle state until the device reaches the end of the kernel execution.

Certainly, CUDA is capable of enabling *asynchronous concurrent execution* between host and device, which returns a control to the host before the device has completed a requested task (i.e., non-blocking). In addition, it is possible to use any other programming languages such as OpenMP and Pthreads to allow host threads to consume the CPU time. However, the concurrent execution gives the control that can only perform operations such as memory copy, setting other input data, and kernel launches using *streams*, and using an additional programming language in addition to CUDA can make harder to design the program structure.

Figure 3 illustrates a result of our motivation experiment, which shows the CPU utilization during application runtime. Since the CUDA applications used in this experiment has very short execution time, we have modified the code of the program by increasing the number of iterations for the kernel launch statement, and the input data size in order to measure the CPU utilization value correctly; we have measured the utilization every 10 milliseconds by using the time stamp counter instruction (i.e., rdtscp). In this experiment, most of CUDA applications uses just one CPU thread for the data initialization and the post-processing, and the CPU thread is held in the idle state during the kernel execution. A few of the CUDA applications, such as `kmeans` from Rodinia benchmark suite [6], is designed such that they use a multithreaded language (e.g., OpenMP) for using multiple CPU threads.

In most of the cases, as a result, the idle time causes an inefficient utilization of the CPU hardware resource of the host machine. However, there is an opportunity to boost CUDA applications if the idle cores participate in the computations of kernels. The execution time could be reduced as much as the participation core count. This paper focuses on allowing the idle cores to cooperate with GPUs for the kernel execution in order to reduce the execution time of CUDA applications (as depicted in Fig. 2b), and thus maximizing system throughput at a given time.

3.2 Problem Analysis

As mentioned briefly, CUDA threads within a single block may access a shared memory space to communicate with each other (i.e., inter-thread synchronization), and threads across multiple thread blocks may access the global memory space (i.e., inter-block synchronization). However, CUDA supports inter-thread synchronization method only. The reason for this restriction is to allow flexibility in the hardware thread scheduler, and to enable the code to be scalable. The only supported inter-block synchronization method is to launch another kernel within the same stream.

In addition, thread blocks within a kernel are required to execute independently. Each of them can get allocated to any available SM so that the GPU can execute them in any order, in parallel or in series. For thread blocks with inter-thread synchronization only, it would be possible to execute thread blocks on any processing unit: CPUs, GPUs, or coprocessors.

The three computational kernels used in this experimental analysis are selected from the CUDA Software Development Kit (SDK) (see Table 1 which also shows the initial execution time for each). For the initial analysis, we have measured the execution delay using only the GPU device and the delay using only the host CPU (through the LLVM JIT compilation technique [7,8,14]). In addition, the workload has been configured either as executing only one thread block or as executing the complete set of thread blocks.

Figure 4a shows the way to find an optimal workload ratio; the x-axis represents the workload ratio in terms of thread blocks assigned to the CPU cores against thread blocks on the GPU device. With having more thread blocks on the CPU cores, fewer thread blocks would be assigned to the GPU device. Therefore, the execution delay for GPU is proportionally reduced along the x-axis. For example, 176:80 as shown in Fig. 4c means that 176 thread blocks are executed on the GPU device, and the remaining 80 thread blocks are assigned to the CPU cores, which means that GPU takes 68.75 % of the workload of the kernel and CPU takes 31.25 %.

From the above analysis, the maximum value between the CPU execution delay and the GPU execution delay at a given workload ratio can be considered as the total execution delay. Thus, each of the crossing points of two lines in Fig. 4 gives the

Table 1 Initial investigation on three CUDA applications

Application	# Thread blocks	Workload	Execution time (ms)		Initial analysis	
			CPU only	GPU only	The best Perf.	Optimal ratio (CPU:GPU)
Binomial options	512	1	17.43	1.15	556.06 (×1.06)	6.2:93.8
		512	8926.6	593.0		
matrixMul	128	1	44.23	12.84	1274.15 (×1.29)	22.5:77.5
		128	5661.8	1644.1		
Transpose	256	1	0.20	0.25	28.81 (×2.19)	55.5:44.8
		256	52.21	64.29		

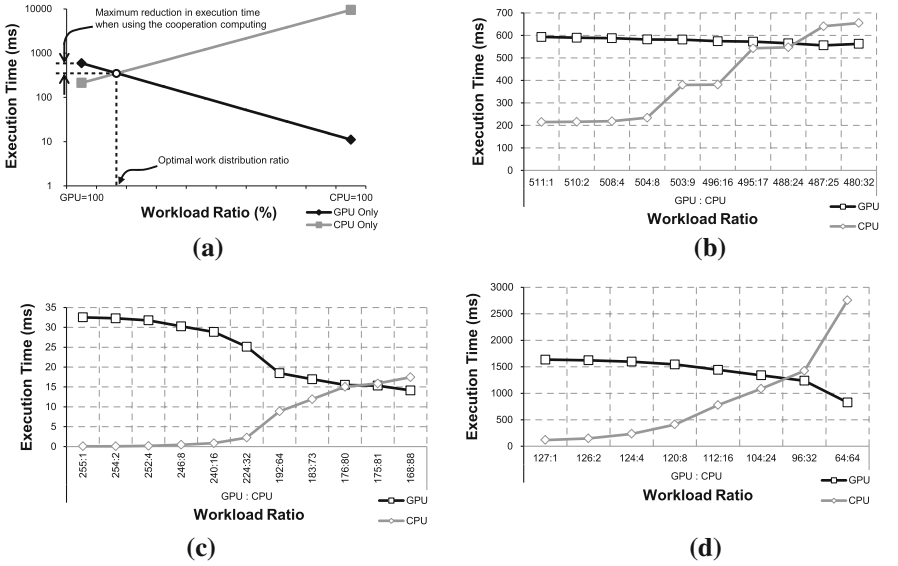

Fig. 4 Initial analysis: **a** a prediction method of theoretical performance improvement with initial execution time **b** Binomial options **c** transpose **d** matrixMul

optimal distribution ratio as well as the maximum performance with the initial data and the mathematical analysis.

4 Software Framework Design

The CHC framework is to use the idle computing resource with concurrent execution of the CUDA kernel on both CPU and GPU (as described in Fig. 2b). An overview of our proposed CHC system is shown in Fig. 5. It contains two runtime procedures for each kernel launched. Each kernel execution undergoes those procedures. The first includes the *Workload Distribution Module* (WDM), designed to apply the distribution ratio to the kernel configuration information. Then, the modified configuration information is delivered to both the *CPU Loader* and the *GPU Loader*. Two sub-kernels ($Kernel_{CPU}$ and $Kernel_{GPU}$) are loaded and executed, based on the modified kernel configurations produced by the WDM.

The second procedure is designed to translate the parallel thread execution code, a virtual instruction set architecture [20], into the LLVM intermediate representation (LLVM IR). As seen in Fig. 5, this procedure extracts the PTX code from the CUDA binary to prepare the LLVM code for cooperative computing. On the GPU device, our runtime system passes the PTX code through the CUDA device driver, which means that the GPU executes the kernel in the original manner using the PTX-JIT compilation. On the CPU core side, CHC uses the PTX translator provided in Ocelot in order to convert PTX instructions into LLVM IR [8]. This LLVM IR is used for a kernel context of all thread blocks running on CPU cores, and LLVM-JIT is utilized to execute the kernel context [14].

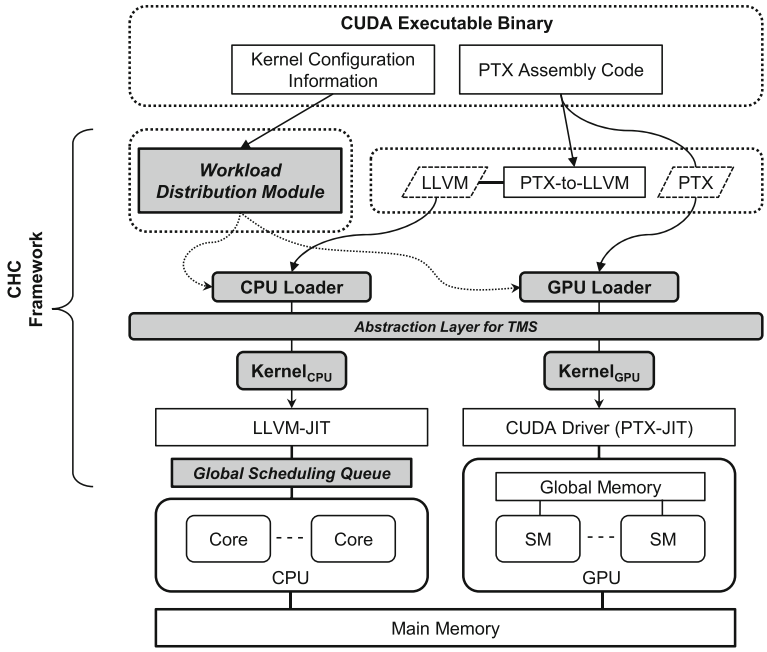

Fig. 5 An overview of the CHC runtime system: the *shaded boxes* represent our implementations

The CUDA kernel execution typically needs some start-up time to initialize the GPU device. In the CHC framework, the GPU start-up process and the PTX-to-LLVM translation are simultaneously performed to hide the PTX-to-LLVM translation overhead.

4.1 Workload Distribution Module and Method

The input of WDM is the kernel configuration information and the output specifies two different portions of the kernel, each for CPU cores and the GPU device. The kernel configuration information contains the execution configuration which provides the dimension of a grid and that of a block. The dimension of a grid can be efficiently used for our workload distribution module.

In order to divide the CUDA kernel, the workload distribution module determines the amount of the thread blocks to be detached from the grid considering the dimension of the grid and the workload distribution ratio as depicted in Fig. 6. As a result, WDM generates two additional execution configurations, one for CPU and the other for GPU. WDM then delivers the generated execution configurations (i.e., the output of the WDM) to the CPU and GPU loaders. With these execution configurations, each loader now can make a sub-kernel by using the kernel context such as LLVM and PTX.

Typically, WDM assigns the front portion of thread blocks to the GPU-side, while the rest is assigned to the CPU-side. Therefore, the first identifier of the CPU's

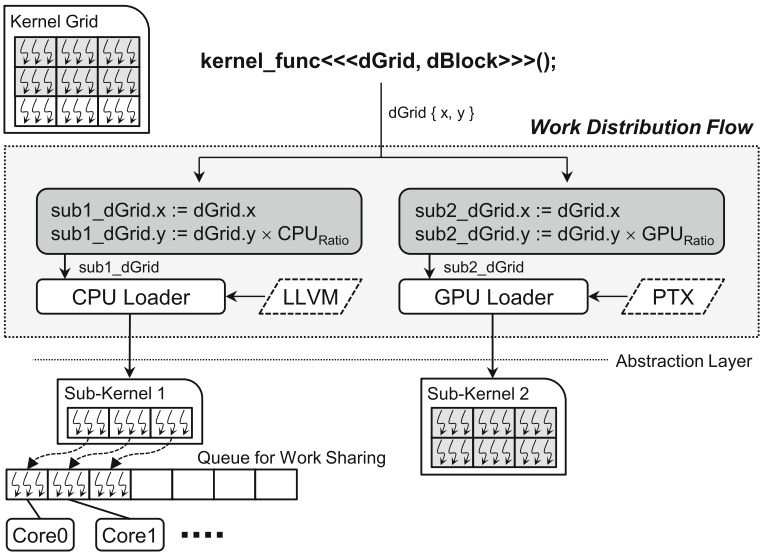

Fig. 6 Work distribution flow and kernel mapping to CPU and GPU

sub-kernel will be $(dGrid.y \times GPU_{Ratio}) + 1$. Then, each thread block can identify the assigned data with the identifier since both sides have an identical memory space.

In order to find the optimal workload distribution ratio, we can probably predict the runtime behavior such as the execution delay on CPU cores. However, it is quite hard to predict characteristics of a CUDA program since the runtime behavior strongly relies on dynamic characteristics of the kernel [2,12]. For this reason, Qilin used an empirical approach to achieve their proposed adaptive mapping [17]. In fact, our proposed CHC also adopts a heuristic approach to determine the workload distribution ratio. Then, the CHC framework performs the dynamic work distribution at runtime based on this ratio. The proposed work distribution can split the kernel according to the granularity of thread block.

4.2 Memory Consolidation for Transparent Memory Space

A programmer writing CUDA applications should assign memory spaces in the device memory of the graphics hardware. These memory locations (or, addresses) are used for the input and output data. In the CUDA model, data can be copied between the host memory and the dedicated memory on the device. For this purpose, the host system should preserve *pointer variables* pointing to the location in the device memory.

As opposed to the original CUDA model, *two* different memory addresses exist for *one* pointer variable in our proposed CHC framework. The key design problem is caused by the fact that the computation results of the CPU side are stored into the main memory that is different from the device memory. To address this problem, we propose and design an abstraction layer, *Transparent Memory Space* (TMS), to preserve two different memory addresses in a pointer variable at a time.

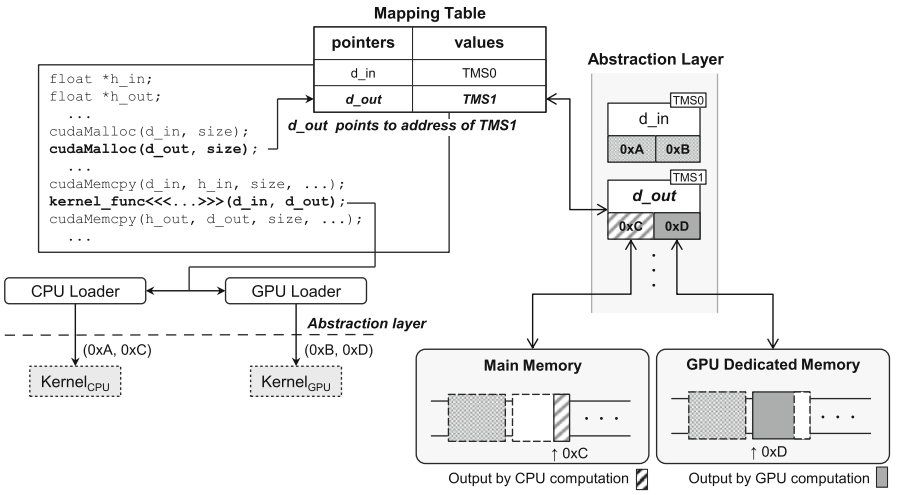

Fig. 7 Anatomy of transparent memory space

4.2.1 Accessing Memory Addresses

The abstraction layer uses *double pointers* data structures (similar to [26]) for pointer variables to map *one* pointer variable onto *two* memory addresses: for the main memory and the device memory. As seen in Fig. 7, we have declared the abstraction layer that manages a list of the TMS data structures. Whenever a pointer variable is referenced, the abstraction layer translates the pointer to the memory addresses, for both CPU and GPU. For example, when a pointer variable (e.g., *d_out*) is used to allocate device memory using `cudaMalloc()`, the framework assigns memory spaces both on the device memory and the host memory. The addresses of these memory spaces are stored in a TMS data structure (e.g., *TMS1*), and the framework maps the pointer variable on the TMS data structure. Thus, the runtime framework can perform the address translation for a pointer variable.

4.2.2 Launching CUDA Kernels

For launching a kernel, pointer variables defined in advance may be used as arguments of the kernel function. At that time, the CPU and GPU loaders obtain each translated address from the mapping table so that each sub-kernel could retain actual addresses on its memory domain.

4.2.3 Merging Separated Data

After finishing the kernel computation, the computation results are copied to the host memory (`cudaMemcpy()`) to perform further operations. Therefore, merging the data of two separate memory domains is required. To reduce memory copy over-

head, the framework traces memory addresses which are modified by the CPU-side computation.

4.3 Global Scheduling Queue for Thread Scheduling

GPU is a throughput-oriented architecture which shows outstanding performance with applications having a large amount of data parallelism [9]. However, to achieve meaningful performance from the CPU side, scheduling thread blocks with an efficient policy is important.

Ocelot uses a locality-aware *static* partitioning scheme in their proposed thread scheduler, which assigns each thread block considering load balancing between neighboring worker threads [8]. However, this static partitioning method probably causes some cores to finish their execution early. In our scheduling scheme, we allow a thread block to be assigned dynamically to any available core. For this purpose, we have implemented a work sharing scheme using a *Global Scheduling Queue* (GSQ) [1,4]. This scheduling algorithm enqueues a task (i.e., a thread block) into a global queue so that any worker thread on an available core can consume the task. Thus, this scheduling scheme allows a worker thread in each core to pick up only one thread block and achieve load balancing. In addition, any core which finishes the assigned thread block so early would handle another thread block without being idle.

5 Experimental Evaluation

The CHC framework has been implemented on a Linux (2.6.28 kernel) system with two-socket Intel Xeon X5550 quad-core processors at 2.66 GHz, 16 GB DDR3 main memory, and an NVIDIA GeForce 9400 GT device. The GPU has 2 SMs, 8 CUDA cores for each SM (total 16 CUDA cores), the GPU clock speed is 1.35 GHz, and 256 MB device memory size. This low-end GPU has a computing capacity that is analogous to that of integrated GPUs. The hardware configuration we have used could ultimately demonstrate the feasibility of the CHC technique on single-chip heterogeneous multicore processors, which have an integrated GPU.

We adapt 14 CUDA applications; twelve from the NVIDIA CUDA SDK [19], SpMV [3], and MD5 hashing [11]. Table 2 summarizes these applications and kernel configurations. From left to right the columns represent the application name, the number of computation kernels, the grid dimension in the kernels, and a brief description of the kernels.

5.1 Performance Improvements

In this experiment, we have measured the execution time of kernel launches and compared CHC framework against the GPU-only computing. All of the validity of the CHC results was compared to computation results that have been executed on a CPU. Figure 8 shows the performance improvements of CHC normalized to GPU-only computations; as expected, the performance of CHC improves in general com-

Table 2 List of test applications and the workload used in evaluation

Application (abbreviation)	# Kernels	Grid dim	Description
3DFD (3DFD)	1	(20 × 20)	3D finite difference computation
Binomial options pricing (BINO)	1	(512 × 1)	European options under binomial model
Black Scholes (BLKS)	1	(480 × 1)	European options by Black-Scholes formula
Mersenne twister (MERT)	2	(32 × 1)	Mersenne twister random number generator and Cartesian Box-Muller transformation
Matrix multiplication (MAT)	1	(128 × 128)	Matrix multiplication: C := A × B
Monte Carlo (MONT)	2	(256 × 1)	European options using Monte Carlo approach
Scalar product (SCALAR)	1	(128 × 1)	Scalar products of input vector pairs
Scan (SCAN)	3	(256 × 1)	Parallel prefix sum
Convolution texture (CONV)	2	(192 × 128)	Image convolution filtering
Transpose (TRANS)	2	(128 × 256)	Matrix transpose
Sobol QRNG (QRNG)	1	(1 × 100)	Sobol's quasi-random number generator
Vector addition (VEC)	1	(196 × 1)	Vector addition: C := A + B
Sparse matrix-vector multiplication (SPMV)	2	(1,024 × 1)	Matrix-vector multiplication: y+ = A × x
MD5 hashing (MD5)	2	(33,312 × 1)	MD5 calculation and search

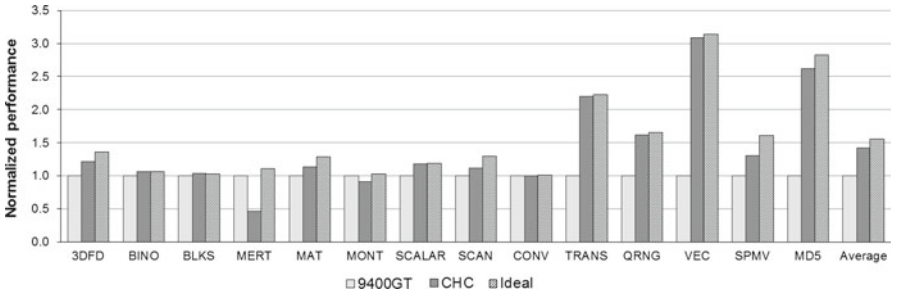

Fig. 8 Normalized performance speedup of CHC over the GPU-only processing

pared to the execution delay using only GPU. The speedup is achieved, ranging from 0.46× for `MERT` up to 3.08× for `VEC`. The average speedup of the CHC framework is 1.42×.

In fact, the applications with exponential, trigonometric, or power arithmetic operations (`BINO`, `BLKS`, `MERT`, `MONT`, and `CONV`) show little performance improvement. In fact, the execution time of these applications on CPU is much higher compared to the execution time on GPU. This is due to the fact that the GPU device normally provides special functional units for those operations. In the case of `MERT`, specifically, the workload of the kernel is so small that launching the kernel on the CPU side has a relatively large cost compared to the GPU, and therefore the speedup is the worst result in all of the tested applications (will be discussed in Sect. 5.4).

Table 3 Initial analysis and CHC results

Abbreviation	Grid dim	Workload	Execution time (ms)		Initial analysis		CHC results	
			CPU only	GPU only	Max. Perf.	Optimal ratio	Actual Perf.	Actual ratio
3DFD	20 × 20	20 × 1	19.468	7.144	104.52	26.8:73.2	117.32	10.0:90.0
		20 × 20	389.36	142.88				
BINO	512 × 1	1 × 1	17.43	1.15	556.06	6.2:93.8	559.44	4.7:95.3
		512 × 1	8926.6	593.0				
BLKS	480 × 1	1 × 1	1.08	0.03	16.98	3.3:96.7	17.02	0.6:99.4
		480 × 1	520.20	17.557				
MERT	32 × 1	1 × 1	13.62	1.55	44.73	10.3:89.7	108.26	3.1:96.9
		32 × 1	435.94	49.85				
MAT	128 × 128	128 × 1	44.23	12.84	1274.15	22.5:77.5	1453.19	18.0:82.0
		128 × 128	5661.8	1644.1				
MONT	256 × 1	1 × 1	8.47	0.28	69.90	3.2:96.8	79.39	3.1:96.9
		256 × 1	2170.1	72.23				
SCALAR	128 × 1	1 × 1	0.22	0.04	4.79	16.6:83.4	4.86	25.0:75.0
		128 × 1	28.86	5.74				
SCAN	256 × 1	1 × 1	0.014	0.004	0.87	23.2:76.8	1.01	9.4:90.6
		256 × 1	3.75	1.13				
CONV	192 × 128	192 × 1	8.45	0.15	19.51	1.8:98.2	19.51	0.8:99.2
		192 × 128	1082.3	19.53				
TRANS	128 × 256	128 × 1	0.20	0.25	28.81	55.5:44.8	29.29	50.0:50.0
		128 × 256	52.21	64.29				
QRNG	1 × 100	1 × 1	0.05	0.01	0.87	21.8:78.2	0.90	16.0:84.0
		1 × 100	5.22	1.46				
VEC	196 × 1	1 × 1	0.005	0.011	0.68	68.5:31.5	0.70	44.9:55.1
		196 × 1	1.01	2.17				
SPMV	1024 × 1	1 × 1	0.003	0.001	1.17	38.2:61.8	1.45	25.0:75.0
		1024 × 1	3.07	1.90				
MD5	33312 × 1	1 × 1	0.002	0.004	52.14	64.7:35.3	56.34	64.0:36.0
		33312 × 1	80.55	147.84				

On the other hand, the applications without those arithmetic operations (`TRANS`, `VEC`, `SPMV`, and `MD5`) show relatively higher speedups due to the simplicity of their operations.

More in detail, Table 3 shows the maximum performance and the optimal distribution ratio obtained from the initial analysis. The actual execution time and the actual work distribution ratio using CHC are also presented; each of the speedup shown in Fig. 8 can be obtained by dividing the execution time on GPU only by the actual CHC performance. In fact, the optimal distribution ratio is used to determine the work distribution ratio on the CHC framework.

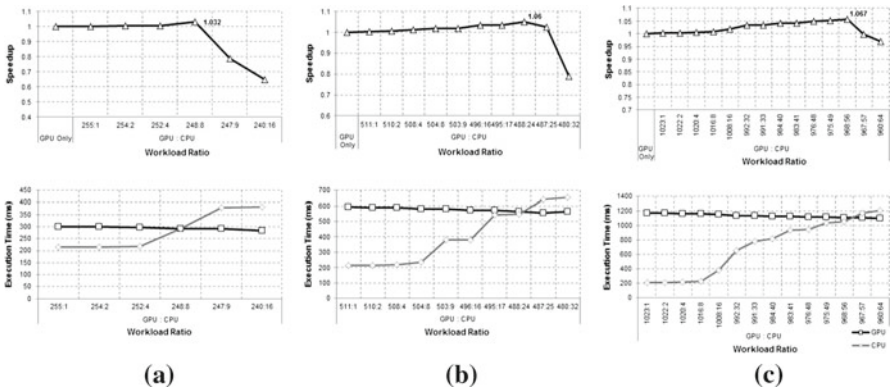

Fig. 9 Impact of the actual distribution ratio on the performance improvement: **a** 256 × 1, **b** 512 × 1, **c** 1,024 × 1

Based on the fourth column, which shows the execution time, we observe that the speed gap between the CPUs and the GPU directly affects the performance improvement. In the case of `BINO` that shows 1.06× speedup, the CPU side is about 15.05× slower than the GPU. In contrast, `TRANS`, `VEC`, and `MD5`, in which the CPU side is quite faster than the GPU even if the kernels are translated to LLVM IR codes, achieve a significant speedup.

5.2 Impact of Workload Distribution Ratio on CHC Performance

We now show the impact of the actual distribution ratio on the performance improvement. The actual distribution ratio for a kernel is obtained from the optimal ratio of the initial analysis. Then, we can assign the portions of thread blocks for the CPU cores to WDM (as described in Sect. 4). Figure 9 shows the speedup and the execution delay of the CPU and the GPU according to the workload distribution ratio. In Fig. 9a, at which the workload (i.e., grid dimension of the kernel) is 256 × 1, the actual performance peaks when the workload distribution ratio of GPU to CPU is 248:8. Interestingly, the experimental speedup is rapidly degraded when the workload of the CPU cores is above eight. The reason for the significant performance degradation is that the number of CPU cores participated in the kernel computation is limited to eight. As mentioned in Sect. 4, a thread block get allocated on a single CPU core, so that execution time on CPU cores is increased significantly whenever the number of thread blocks is a multiple of the CPU core count. In Fig 9b, the runtime behavior due to the CPU core count is observed more in detail. There is a substantial increase in execution time according to the workload of CPU (from 8 to 9, from 16 to 17, and from 24 to 25). As a result, this runtime behavior implies that the maximum experimental speedup is generally obtained when the number of CPU cores is a multiple of eight, which means the GSQ efficiently distributes thread blocks to each of CPU cores.

Figure 10 shows a comparison between the ideal performance obtained from the theoretical analysis and the actual results. On average, experimental results on actual

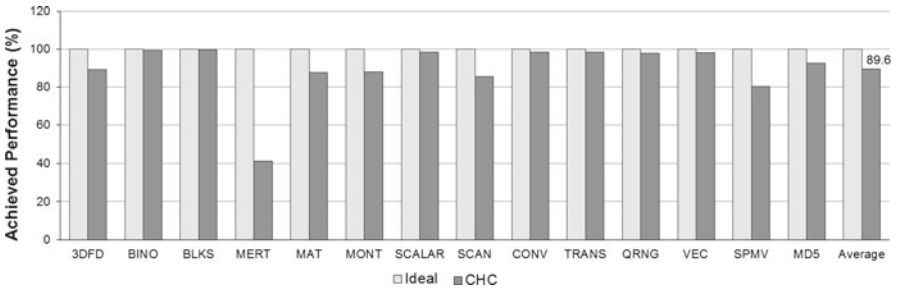

Fig. 10 Comparison between the optimal performance improvements and experimental results

machines show a performance which is 89.6 % of the theoretically predicted performance. This demonstrates that the proposed framework successfully utilizes the CPU resource for the CUDA kernels.

5.3 Impact of Workload Characteristics

The PTX code which corresponds to a CUDA kernel has a great impact on the performance of the CHC runtime system. The runtime system extracts PTX instructions of the kernel from the CUDA binary to generate the corresponding LLVM IR code. Since the LLVM IR code corresponds to the computational workload for the CPU side, the execution time of the workload is highly dependent on the computational complexity of the LLVM IR code; therefore, the performance of the kernel execution essentially depends on the complexity of the original PTX code.

We have investigated the PTX assembly codes of all the CUDA applications to determine which types of PTX instructions are the dominant factors in the runtime performance of the CHC framework. To this end, we classified the PTX instructions according to their *opcodes*, into eight types of instructions: integer arithmetic, floating point, special function (`cos`, `sin`, `exp`, and `log`), logic and shift, load, store, data movement, control, and synchronization instructions. For example, `add` is an integer arithmetic instruction that operates on the integer types in register, and `cos` that finds the cosine of a value is a special instruction, exploited by special function units (SFUs) on GPUs.

Figure 11a presents the PTX instruction breakdown and the speedups for the previous results in Sect. 5.1 (Fig. 11b on average). Each bar shows the fraction of PTX instructions for a CUDA application, and the line across all bars represents the normalized performance improvements for all the CUDA applications. For those kernels in which the cooperative execution performs well, the number of floating-point arithmetic and store instructions is relatively small. `VEC`, which shows the best performance improvement, has three dominant types of the instructions: 28.6 % integer arithmetic, 28.6 % load, 19.0 % data movement instructions, whereas floating-point arithmetic, special, and store instructions show the same value of 4.8 %. On the other hand, `BLKS` and `MONT`, which show poor performance gains, have a significant portion of floating-point instructions (36.7 and 25.1 %, respectively). This suggests that floating-point

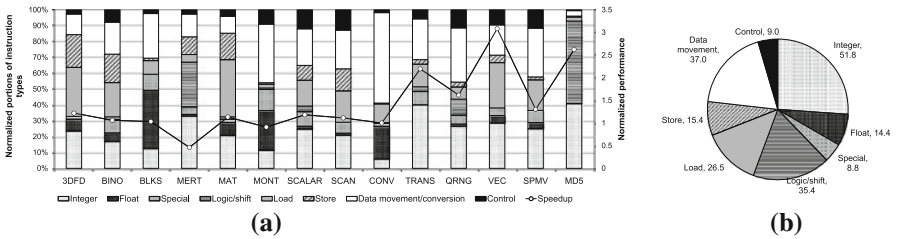

Fig. 11 The fraction of PTX instructions: **a** PTX instruction breakdown and speedups of the CUDA applications **b** on average (in percent)

instructions directly impact the runtime performance on the CPUs due to the fact that the theoretical upper bound on 32-bit floating-point multiply-accumulate instructions (e.g., `fma`) is one-third times lower than that of 32-bit integer instructions (in addition, special instruction throughput are two orders of magnitude lower than `fma`) [8]. Attacking this challenge in translation efficiency will be a part of our future work. It may be overcome with an advanced compilation technique (see Sect. 6).

5.4 Composition Effect of PTX Instructions

We now describe the effect of the PTX instruction count according to their types. The normalized fraction of PTX instruction types shows only the relative portions of the instructions between the kernels and does not represent the effect of the actual number of the instructions on the runtime performance. For example, `BLKS` has 47 floating-point instructions, 36.7 % of the total number of PTX instructions, while `MONT` has 53 floating-point instructions, 25.1 % of the total instructions; however, `BLKS` shows better performance improvement (1.03×) than `MONT` (0.9×) due to a smaller instruction count. This introduces an incremental experiment that evaluates the effect of individual types of PTX instructions (Fig. 12).

We find that the upper bound of the CHC throughput is generally limited by memory access latency. In case of GPUs, TLP can effectively handle and hide most memory access latencies since GPU cores have significantly higher TLP. On the other hand, CPU cores which have lower TLP hardly run the translated LLVM IR codes due to the massive data parallelism. From both Fig. 12b, c, we can see a tendency for a lower performance improvement as the number of load/store instructions increases.

In addition, Fig. 12e presents the ratio of store instructions to load instructions for all the kernels. Interestingly, `MERT`, the worst case, shows a remarkable ratio of 2.3; however, the others are lower than 1. This implies that MERT is not suitable to run on CPUs since the impact of memory access latencies on the performance is more significant as store instructions are more frequently executed.

For each kernel, we also define an aggregate amount of the following three numbers: the floating-point instruction portions of all the PTX instructions, the special instruction portion, and the ratio of store instructions to load instructions. Figure 12f illustrates the composition effect of the three instruction types on the CHC runtime performance and helps us understand the interaction among PTX instructions more clearly. Based

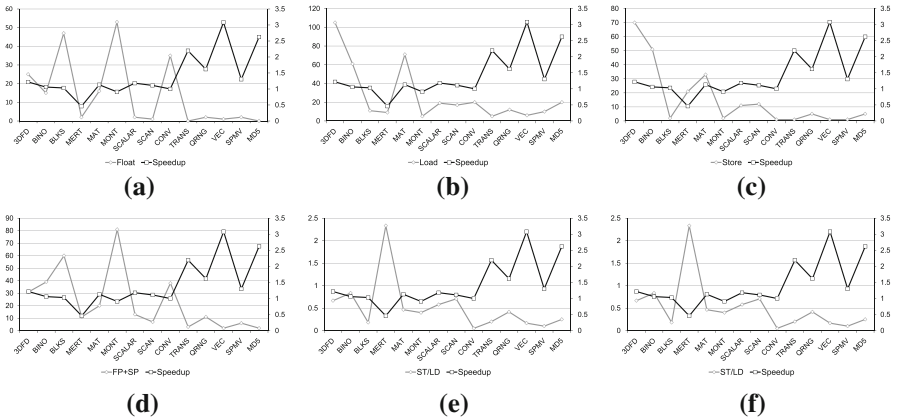

Fig. 12 The number of PTX instructions according to their types: **a** floating-point, **b** load, **c** store, **d** the sum of floating-point and special instructions **e** the ratio of store instructions to load instructions **f** putting all in together

on this observation, the efficiency of the proposed cooperative computing for CUDA applications is much more predictable. Furthermore, the CUDA applications can be either defined as CPU-friendly applications or GPU-friendly applications. For example, `MERT`, which has the highest score, is a clearly GPU-friendly workload and is preferable to GPU-only computing, whereas `TRANS`, `QRNG`, `VEC`, and `MD5` have CPU-friendly characteristics. The remaining kernels, in which the aggregate values are within from 0.5 to 1.0, may have an opportunity to be designed and implemented either for CPUs, GPUs, or CPU–GPU hybrid configurations.

6 Discussions

The CUDA architecture supplies a transparency of application software such as language, compiler, driver, and useful tools. This abstraction layer allows developers to program their applications without any knowledge on hardware architecture. However, the CPU that generally controls an execution flow of CUDA applications should wait for its turn during the kernel execution. This paper proposes a runtime CUDA framework, which enables CUDA applications running on GPU to be concurrently executed on multicore CPU and GPU. There are two topics that will be discussed in this section.

6.1 PTX Translation to LLVM

The runtime system uses Ocelot to translate PTX instruction to LLVM IR. The number of LLVM instructions per a PTX instruction (i.e., translation efficiency) varies from one to tens, so that the number of the translated LLVM instructions is important for the performance. From the hardware perspective, the quad-core CPU within the system used in Sect. 5 is manufactured in the Nehalem architecture, which has the execution

engine that carries out four single-precision (SP) floating-point operations (FLOPs) such as addition or subtraction. Therefore, the ideal throughput of the Nehalem core is 10.64 Giga FLOPs/sec/core (2.66 GHz $\times$ 4 FLOPs/Hz), and the whole throughput of the two-socket system is then 85.12 GFLOPs. On the contrary, the NVIDIA GeForce 9400 GT has 67.2 GFLOPs in terms of single-precision FP operations. Although the performance of the CPU is better than that of the GPU, the performance improvement can be significantly varied according to the translation efficiency. Thus, if we can map PTX instructions onto LLVM instructions more efficiently with an advanced compiler technique such as a PTX to AVX native translation, then more speedup can be achieved.

6.2 Global Memory Consistency

Cooperative heterogeneous computing emulates the global memory on the CPU-side. Thread blocks in the CPU can access the emulated global memory and perform the atomic operations. However, our system does not allow the global memory atomic operations between the thread blocks on the CPU and the thread blocks on the GPU to avoid severe performance degradation. In fact, discrete GPUs have their own memory and communicate with the main memory through the PCI express, which causes long latency problems. This architectural limit suggests that the CHC prototype need not provide global memory atomic operations between CPU and GPU. To allow the atomic operations without performance degradation, a system with an integrated GPU, in which CPU and GPU share a memory space, or a new hardware architectural support is required.

Besides the CUDA architecture, the key difference on which we focus is in the use of idle computing resources with a concurrent execution of the same CUDA kernel on both CPU and GPU, thereby finding a new way of computing on a heterogeneous platform that consists of CPU and GPU. Considering that the future computer systems are expected to incorporate more cores in both general purpose processors and graphics devices, parallel processing on CPU and GPU would become a great computing paradigm for high-performance applications. This would be quite helpful to programs on a single chip heterogeneous multi-core processor including CPU and GPU as well. Note that Intel and AMD have already shipped commercial heterogeneous multi-core processors.

7 Related Work

There have been several prior research projects which aim at mapping an explicitly parallel program for graphics devices onto multi-core CPUs or heterogeneous architectures. OpenCL [21] is an open standard for parallel programming that allows developers to conveniently access to CPUs, GPUs, and other processing units. The kernels can be assigned to any processing unit and executed simultaneously. However, there is a significant difference between OpenCL and our proposed framework which is kernel partitioning. In fact, the CHC system has finer granularity than OpenCL, dividing a single kernel into multiple pieces for both CPUs and GPUs. The OpenCL

standard makes no consideration for kernel partitioning. MCUDA [25] automatically translates CUDA codes for general purpose multi-core processors, applying source-to-source translation. This implies that the MCUDA technique translates the kernel source code into a code written in a general purpose high-level language, which requires one additional step of source recompilation.

Twin Peaks [10] maps an OpenCL-compatible program targeted for GPUs onto multi-core CPUs by using the LLVM (Low Level Virtual Machine) intermediate representation for various instruction sets. Ocelot [8], which inspired our runtime system, uses a dynamic translation technique to map a CUDA program onto multi-core CPUs. Ocelot converts at runtime PTX code into an LLVM code without recompilation and optimizes PTX and LLVM code for execution by the CPU. The proposed framework in this paper is largely different from these translation techniques (MCUDA, Twin Peaks, and Ocelot) in that we support cooperative execution for parallel processing over both CPU cores and GPU cores.

In addition, EXOCHI provides a programming environment that enhances computing performance for media kernels on multicore CPUs with Intel Graphics Media Accelerator (GMA) [27]. However, this programming model uses the CPU cores only for serial execution. The Merge framework has extended EXOCHI for the parallel execution on CPU and GMA; however, it still requires APIs and the additional porting time [16]. Saha et al. [23] have proposed a programming model for a different platform, which consists of x86 CPU and Larrabee cores [24]. Lee et al. [15] have presented a framework which aims at porting an OpenCL program on the Cell BE processor to manage software-managed caches and coherence protocols.

Ravi et al. [22] have proposed a compiler and a runtime framework that generate a hybrid code running on both CPU and GPU. It dynamically distributes the workload, but the framework targets only for generalized reduction applications, while our system targets to map general CUDA applications. Qilin [17], in the most relevant study to our proposed framework, has shown an adaptive kernel mapping using a dynamic work distribution. The Qilin system trains a program to maintain databases for the adaptive mapping scheme. In fact, Qilin requires and strongly relies on its own programming interface. This implies that the system cannot directly port the existing CUDA codes, but rather programmers should modify the source code to fit their interfaces. As an alternative, CHC is designed for seamless porting of the existing CUDA code on CPU cores and GPU cores. In other words, we focus on providing backward compatibility of CUDA runtime APIs.

8 Conclusions

The paper has introduced three key features for the efficient exploitation of the thread level parallelism provided by CUDA on the CPU multi-cores in addition to the GPU device. The proposed CHC framework provides a tool set which enables CUDA binary to run on CPU and GPU, without imposing source recompilation. The experiments demonstrate that the proposed framework successfully achieves efficient parallel execution and that the performance results obtained are close to the values deduced from the theoretical analysis. We believe the CHC can be utilized in the future heteroge-

neous multi-core processors which are expected to include even more GPU cores as well as CPU cores.

As future work, we will first develop a dynamic control scheme on deciding the workload distribution ratio. We also plan to design an efficient thread block distribution technique considering data access patterns and thread divergence. We believe CHC can eventually provide a solution for the degradation of performance due to the irregular memory access and thread divergence in the original CUDA execution model. In fact, the future CHC framework needs to address the performance trade-offs considering the CUDA application configurations on various GPU and CPU models. In addition, we will discuss the overall speedups considering the transition overhead to find the optimal configuration for the CHC execution model.

Acknowledgments We thank all of the anonymous reviewers for their comments. This work was supported by the Basic Science Research Program through the National Research Foundation (NRF) of Korea, which is funded by the Ministry of Education, Science and Technology [2009-0070364]. This work is also supported in part by the US National Science Foundation under Grant No. CCF-1065448. Any opinions, findings, and conclusions as well as recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '00, pp. 1–12. ACM, New York, NY, USA (2000)
2. Bakhoda, A., Yuan, G., Fung, W., Wong, H., Aamodt, T.: Analyzing cuda workloads using a detailed gpu simulator. In: Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on, pp. 163–174 (2009). doi:10.1109/ISPASS.2009.4919648
3. Bell, N., Garland, M.: Cusp: Generic parallel algorithms for sparse matrix and graph computations (2010). http://cusp-library.googlecode.com
4. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**, 720–748 (1999)
5. Brodtkorb, A.R., Dyken, C., Hagen, T.R., Hjelmervik, J.M., Storaasli, O.O.: State-of-the-art in heterogeneous computing. Sci. Program. **18**, 1–33 (2010)
6. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.H., Skadron, K.: Rodinia: a benchmark suite for heterogeneous computing. In: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, pp. 44–54 (2009). doi:10.1109/IISWC.2009.5306797
7. Cifuentes, C., Malhotra, V.M.: Binary translation: static, dynamic, retargetable? In: Proceedings of the 1996 International Conference on Software Maintenance, ICSM '96, pp. 340–349. IEEE Computer Society, Washington, DC, USA (1996)
8. Diamos, G.F., Kerr, A.R., Yalamanchili, S., Clark, N.: Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10, pp. 353–364. ACM, New York, NY, USA (2010)
9. Garland, M., Kirk, D.B.: Understanding throughput-oriented architectures. Commun. ACM **53**, 58–66 (2010)
10. Gummaraju, J., Morichetti, L., Houston, M., Sander, B., Gaster, B.R., Zheng, B.: Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10, pp. 205–216. ACM, New York, NY, USA (2010)
11. Juric, M.: Cuda md5 hashing. http://majuric.org/software/cudamd5
12. Kerr, A., Diamos, G., Yalamanchili, S.: A characterization and analysis of ptx kernels. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), IISWC '09, pp. 3–12. IEEE Computer Society, Washington, DC, USA (2009)

13. Kumar, R., Tullsen, D., Jouppi, N., Ranganathan, P.: Heterogeneous chip multiprocessors. Computer **38**(11), 32–38 (2005)
14. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO '04, pp. 75. IEEE Computer Society, Washington, DC, USA (2004)
15. Lee, J., Kim, J., Seo, S., Kim, S., Park, J., Kim, H., Dao, T.T., Cho, Y., Seo, S.J., Lee, S.H., Cho, S.M., Song, H.J., Suh, S.B., Choi, J.D.: An opencl framework for heterogeneous multicores with local memory. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10, pp. 193–204. ACM, New York, NY, USA (2010)
16. Linderman, M.D., Collins, J.D., Wang, H., Meng, T.H.: Merge: a programming model for heterogeneous multi-core systems. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pp. 287–296. ACM, New York, NY, USA (2008)
17. Luk, C.K., Hong, S., Kim, H.: Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on, pp. 45–55 (2009)
18. Nickolls, J., Dally, W.: The gpu computing era. Micro IEEE **30**(2), 56–69 (2010)
19. NVIDIA: Cuda parallel computing platform. http://developer.nvidia.com/category/zone/cuda-zone
20. NVIDIA: Ptx: Parallel thread execution isa. http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation
21. OpenCL: The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl
22. Ravi, V.T., Ma, W., Chiu, D., Agrawal, G.: Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In: Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10, pp. 137–146. ACM, New York, NY, USA (2010)
23. Saha, B., Zhou, X., Chen, H., Gao, Y., Yan, S., Rajagopalan, M., Fang, J., Zhang, P., Ronen, R., Mendelson, A.: Programming model for a heterogeneous x86 platform. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, pp. 431–440. ACM, New York, NY, USA (2009)
24. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: a many-core x86 architecture for visual computing. In: ACM SIGGRAPH 2008 papers, SIGGRAPH '08, pp. 18:1–18:15. ACM, New York, NY, USA (2008)
25. Stratton, J., Stone, S., Hwu, W.m.: Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In: Amaral, J. (ed.) Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, vol. 5335, pp. 16–30. Springer, Berlin (2008)
26. Tian, C., Feng, M., Gupta, R.: Supporting speculative parallelization in the presence of dynamic data structures. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, pp. 62–73. ACM, New York, NY, USA (2010)
27. Wang, P.H., Collins, J.D., Chinya, G.N., Jiang, H., Tian, X., Girkar, M., Yang, N.Y., Lueh, G.Y., Wang, H.: Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, pp. 156–166. ACM, New York, NY, USA (2007)